# Filter Joins: Cost-Based Optimization for Magic Sets

**Praveen Seshadri** **Joseph M. Hellerstein** **Raghu Ramakrishnan**

Computer Sciences Department
U.Wisconsin, Madison WI
*praveen,joey,raghu@cs.wisc.edu*

June 8, 1995

## Abstract

Complex decision-support queries often involve table expressions and views. The current state-of-the-art optimization technique for such queries is the magic sets rewriting. Unfortunately, this rewriting is typically applied as a heuristic query transformation, rather than as part of a cost-based query optimizer. There are some cases in which the rewriting can improve query execution time, and other cases in which it can degrade performance. Further, there are a large number of variants of magic sets rewriting that may be applied to a single query. The optimal choice of how and when to apply magic sets rewriting has remained an open research problem.

In this paper we take a new approach, treating magic sets rewriting as a particular case of a new join algorithm that we call the Filter Join. Adding the Filter Join algorithm to a traditional cost-based query optimizer is difficult, because it can cause an unacceptable increase in the complexity of optimization. We demonstrate how this problem arises, and how it can be addressed by placing reasonable limits on the search space. Using the cost formulas that we derive, an optimizer can examine the Filter Join as a join algorithm option, without adversely affecting the complexity of optimization. This allows magic sets rewriting and similar techniques to be fully integrated with a cost-based query optimizer.

The Filter Join allows the DBMS to find better evaluation algorithms for complex queries. It is also applicable to remote relations in distributed databases, and to relations defined by user functions. We treat all such relations uniformly as "virtual" relations. Any join expression involving virtual relations benefits from the cost-based exploration of a Filter Join. This is especially important for heterogeneous queries involving remote views.

# 1 Introduction

Current relational database systems can treat table expressions and views as relations in an SQL query. Further, in distributed and heterogeneous databases, a remote relation or relational view can be treated as a single relation in a query. Because such relations are not materialized in the (local) database, we call them "virtual" relations. Access to such relations usually contributes significantly to the cost of query evaluation. In this paper, we study the optimization of join queries involving virtual relations.

There are two reasons why this work is relevant and topical. First, there is an increasing interest in processing complex decision-support queries, usually involving views and table expressions. It is, therefore, becoming imperative to optimize the evaluation of such complex queries that have "virtual" relations. Second, complex queries can naturally arise in heterogeneous databases. Previous research on heterogeneity has focused mainly on issues of integration semantics and architecture. However, heterogeneous query processing presents an important challenge: not only could a query access a remote relation, it could even involve a join with a remote view.

Our contributions in this paper are two-fold:

- We introduce the Filter join as a basic join algorithm that should be considered by a query optimizer. We demonstrate that existing heuristic query rewrite techniques (like magic sets) and existing heuristic join techniques (like semi-join) are instances of the Filter join.
- There are obvious situations in which a Filter join is the join algorithm of choice. While this has been observed before in various contexts, we demonstrate that a traditional cost-based optimizer can identify such situations by the use of suitable cost formulas. Further, the asymptotic complexity of the optimization process is unaffected. It is therefore possible to fully incorporate magic sets and similar techniques into a cost-based query optimizer.

Section 2 uses an example to explain the motivation for this work and the issues involved. It also introduces the Filter join method. Section 3 describes a traditional "System-R" optimizer, and the complexity involved in adding this new join method to the optimizer. Section 4 describes the cost-formulas for the join method, and how the components of the cost formulas can be computed. For the sake of clarity, much of the paper is presented in the context of optimizing complex SQL queries that use views. However, we should stress that the ideas presented are directly applicable to joins in heterogeneous databases, and a number of other scenarios as well. Section 5 discusses these issues in some detail. We conclude with a review of related work in Section 6 and a summary of our contributions in Section 7.

# 2 Motivation

We use an SQL query with views in Figure 1 as a motivating example. The query finds every young employee in a big department whose salary is higher than the average salary in that department. The query involves a join between the Emp, Dept and DepAvgSal relations; in this example, the view DepAvgSal is a virtual relation.

Magic sets rewriting is a heuristic technique which has been suggested as a means of optimizing complex queries involving views and table expressions. In some cases, it has been shown to result in orders of magnitude improvement in execution efficiency [MFPR90]. We now demonstrate the result of applying the magic sets rewriting to the original query specification of Figure 1. The purpose of the rewriting is to optimize the join with the view DepAvgSal. The underlying observation is that the average departmental salary need not be computed

```
CREATE VIEW DepAvgSal AS
    (SELECT E.did, AVG(E.sal) AS avgsal
    FROM Emp E
    GROUPBY E.did);
```

```
SELECT E.eid, E.sal, V.avgsal
FROM Emp E, Dept D, DepAvgSal V
WHERE E.did = D.did
    E.did = V.did AND E.sal > V.avgsal AND
    AND E.age < 30 AND
    AND D.budget > 100,000
```

Figure 1: Original Query

```
CREATE VIEW PartialResult AS
   (SELECT E.eid, E.sal, E.did
   FROM Emp E, Dept D
   WHERE E.did = D.did AND E.age < 30 AND
       AND D.budget > 100,000
```

```
CREATE VIEW Filter AS
   (SELECT DISTINCT P.did
   FROM PartialResult P );
```

```
CREATE VIEW RestrictedDepAvgSal AS
   (SELECT F.did, AVG(E.sal) as avgsal
   FROM Filter F, Emp E
   WHERE E.did = F.did
   GROUPBY F.did);
```

```
SELECT P.eid, P.sal, V.avgsal
FROM PartialResult P, RestrictedDepAvgSal V
WHERE P.did = V.did
    AND P.sal > V.avgsal
```

Figure 2: Magic Sets Rewriting

for every department; it need only be computed for those departments that are big and have young employees. If there are few such departments, it is probably desirable to apply the rewriting. The rewritten query (the result of magic sets) is shown in Figure 2.

The PartialResult table represents the partial computation in the main query block at the stage when the view DepAvgSal is joined. From this PartialResult table, a filter set is created, which identifies all those departments for which the average salary needs to be computed. This filter set is now used to restrict the computation of the view[1]. A modified version of the view is created which includes a join with the filter set (thereby restricting the computation in the view to the desired departments). Finally, the restricted view is joined with the PartialResult table to produce the answer.

Since the virtual relation DepAvgSal is joined after first restricting it with a unique filter set, we call this a "Filter" join. Traditionally, magic-sets rewriting has been viewed as query transformation. By treating it as a join technique, as we do in this paper, we can integrate it with existing optimizer technology for evaluating join algorithms.

## 2.1 Optimization Problems

In the rewritten query, the filter set contains all departments which are big and have young employees. This is the most restrictive filter set possible. However, the use of a less restrictive filter set is also correct. As a trivial example, the original query in Figure 1 does not restrict the view at all and it is certainly correct. Similarly,

[1] The filter set has usually been called the "magic" set; hence the name "magic sets rewriting". We use the name "filter set" because it is more meaningful, and also because we are describing the "Filter" join method.

the filter set can contain only the big departments, or only the departments with young employees. While these options may result in more computation inside the view, they could be cheaper overall (because the PartialResult table or the FilterSet is cheaper to create). Finally, if every department is big and has young employees, the rewritten query does not provide any improvement over the original query. On the other hand, it may be more expensive to execute because the PartialResult relation may have to be materialized, the filter set has to be uniquely projected, and the filter set has to be joined in the modified view definition.

There are a few other choices that arise in the context of magic sets rewriting, which this example does not illustrate. When there are multiple join attributes, a choice needs to be made if all the join attributes will contribute to the filter set, or whether only some of the attributes will be used. Further, if there are multiple views in a query, some decision needs to be made regarding their interaction. For instance, if Emp itself were really a view, should Emp be used to generate a filter set for DepAvgSal, or vice-versa, or should only stored relations be used to generate filter sets?

To summarize the above discussion:

- There are cases in which it is undesirable to perform magic sets rewriting.
- When magic sets rewriting is performed, there are a number of possible ways in which the filter set could be created. Each option may be optimal under certain circumstances.
- When there are multiple join attributes, the filter set could contain any subset of them.
- When there are multiple views joined in the query, further decisions need to be made.

In the extensive literature on magic sets rewriting, all of these decisions have been collectively called the "sideways information passing strategy" or "SIPS"; the filter set passes information (on the values of join attributes) "sideways" into the view definition. The choice of exactly what information should be passed is defined by the SIPS. While this is a reasonable abstraction to help understand the problem, no satisfactory solution currently exists for choosing the SIPS in a cost-based manner. Instead, existing systems that perform magic sets rewriting (both in relational and in deductive database systems) have chosen one of two approaches:

- Leave it to the user to specify whether or not to perform magic sets rewriting. If magic sets is performed, the user needs to define the SIPS. This approach is used in CORAL [RSSS93] and is effectively an admission that the problem is too difficult to be tackled by a query optimizer.
- Independently optimize the query with and without magic rewriting and choose the cheaper plan. For magic rewriting, choose a SIPS based on some heuristic. This approach is used in Starburst [MP94]. The SIPS chosen corresponds to the join order that arises from optimizing the original query without magic rewriting. No cost-based justification exists for this heuristic, nor is there any guarantee that an optimal plan is chosen.

## 2.2 Magic Sets Rewriting as a Filter Join

We define the Filter join of relations A and B as follows:

**Definition 2.1 (A Filter –Join B)** A distinct set of values of the join attribute of A is created. This set is used as a filter to restrict the tuples of B that are accessed. This restricted set of B tuples is then joined with the relation A (using any other available join algorithm). A is called the *outer* relation and B is called the *inner* relation in the Filter join.

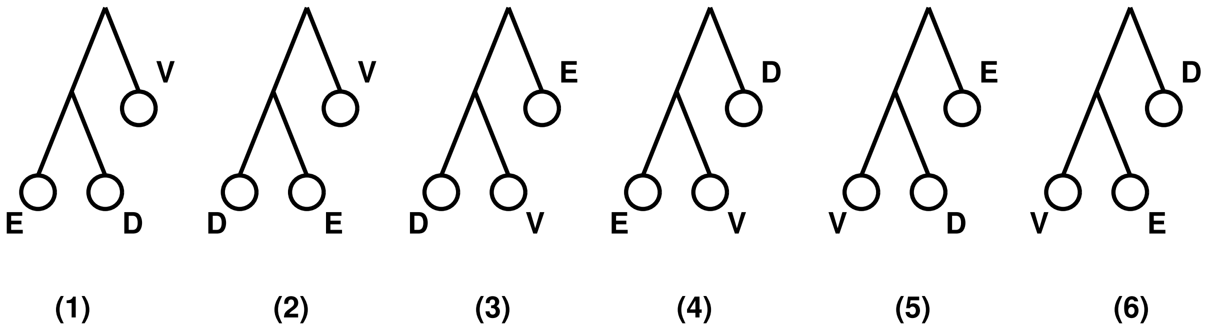


Figure 3: Possible Join Orders

This definition looks similar to the well-known semi-join [BGWRR81] operation, and it has indeed been observed in the past that magic rewriting and semi-joins are similar in spirit. The important distinction that has remained is that magic sets is viewed as a high-level query transformation that can deal with views and table expressions. On the other hand, semi-joins are viewed as a lower level join optimization. Our observation is that both techniques (as well as others: see Section 5) are really instances of the Filter join method. A similar observation has recently been made independently by [SSS95].

Our approach treats magic sets rewriting as a query planning choice that includes a Filter join, with the view being the inner relation. The choice of SIPS then becomes similar to a choice of join order. We now briefly explain the broad ideas; subsequent sections expand on them at some length. For this section, we postpone the issues of multiple join attributes, and multiple views in the same query.

To better understand the interaction between Filter joins and join ordering, let us once again consider the original query of Figure 1. There are six possible choices of join order for the join involving Emp E, Dept D, and DepAvgSal V. These join orders are shown pictorially in Figure 3. When this is related to the magic sets rewriting example of Figure 2, it should be evident that the particular rewriting shown is based on join order 1 or 2. These are the only two join orders that correspond to the PartialResult table generated.

However, join orders 3 and 4 are also perfectly reasonable plans for the query. If magic sets rewriting is based on join order 3, the filter set will contain only the big departments. The actual rewritten query will be textually different from the query in Figure 2. If the rewriting is based on join order 4, the filter set contains the departments with young employees. This results once again in a different rewritten query. Finally, join orders 5 and 6 correspond to the original query (i.e. magic sets rewriting is not performed).

The problem with the magic rewriting technique is that there is a large space of possible variants, and a query rewriting mechanism is not equipped to search this space of options in a cost-based manner. On the other hand, the task of a query optimizer is to explore a space of available plan options. By introducing the Filter join method, and by giving it reasonable cost formulas, we can incorporate it into existing query optimizers.

## 2.3 Optimizing the Filter Join

The traditional query optimization problem is to find a good join order (and corresponding join methods) for a query of multiple relations. The usual assumptions are that the access cost and cardinality of each relation are known, and the selectivities of all predicates are either known or estimated, and are independent. The combination of virtual relations and the Filter join method result in two important variations from these assumptions:

- When a virtual relation is being joined using the Filter join method, its access cost and cardinality are not fixed. They depend critically on the filter set used.
- Given a particular choice of filter set, it may not be cheap to compute these statistics. If the virtual relation is a complex view, it could require a recursive invocation of the optimizer on the view to estimate the statistics.

This supplies the motivation for our work: to devise cost formulas for this new join method, and to incorporate them into a query optimizer without severely compromising the optimizer's efficiency.

# 3 Join Order Optimization

In this section, we describe the most commonly used join optimization algorithm, and describe how the Filter join algorithm described here can be incorporated into the optimizer. A detailed cost formula skeleton is devised for the Filter join. Finally, we look at various approximate solutions to the problem. We note that while our description is in terms of a specific optimizer, the same ideas carry over to other optimization algorithms as well.

A query optimizer considers the problem of determining the optimal order in which to execute the join of N relations. The optimizer also decides on the actual join method used to execute each join (we assume some small constant number of possible join methods considered for each join). An exhaustive search algorithm would be exponentially expensive, because each of the $O((2(N-1))!/(N-1)!)$ options would have to be considered (see [GHK92] for a derivation of the complexity of exhaustive join optimization). This is a prohibitive cost for even a small value of $N$.

## 3.1 System R Optimizer

Many optimizers restrict themselves to plans that are shaped like a *left deep* tree or comb. That is, the inner relation for a join is never an intermediate result of a join, but is always one of the original relations. This restricts the search space (and the complexity of a brute force exhaustive search) to $O(N!)$. In order to find the optimal left deep plan, the System R optimizer [SACLP79] uses a dynamic programming algorithm, which further reduces the complexity.

It starts by considering each of the $N$ relations as the outermost relation in the join plan. For each of them, it tries all possible joins with some other relation. For each particular join of two relations, various join methods are considered. Each join method is associated with a cost formula, and the method that is the least expensive is retained. At the end of this step, the optimal plan has been computed for every join of two relations. When a join method results in a specific physical property (like sort ordering) that might be useful for some future operation, it is not pruned from the set of partial plans. The step is now repeated to generate plans for joins of three relations, and so on, until the optimal plan for the entire join of N relations is determined. The complexity of this algorithm is $O(N * 2^{(N-1)})$, which though exponential, is still better than $O(N!)$. Other algorithms have been suggested that assume certain properties about the join cost formulas and the acyclicity of the join predicates, and actually perform the optimization in polynomial time [IK84, KBZ86]. However, the basic approach of using left deep trees remains the same.

### 3.2 Problems in Costing a Filter Join

Assume that a join involving $N$ relations is being optimized by a System R optimizer augmented to consider Filter joins. At some intermediate stage of the algorithm, it tries to determine the cost of a Filter join. The outer relation is a composite relation of the form $(R_1 \bowtie R_2 ... \bowtie R_{(k-1)})$, and the inner is the virtual relation $R_k$. The most restrictive Filter join possible would create the filter set from the relation $R_{most} = (R_1 \bowtie R_2 ... \bowtie R_{(k-1)} \bowtie R_{(k+1)} ... \bowtie R_N)$. If computing $R_{most}$ were free, and if $R_k$ were the only virtual relation in the query, this might even be the desirable strategy. Less restrictive filter sets could be created from any subset of $R_{most}$; we will call the chosen subset the "*production*" set. The filter set can be represented exactly, or in a lossy fashion (i.e. some superset of the filter set can be used instead). The lossiness may be introduced by an implementation like a Bloom filter, or simply by omitting one of the join attributes.

Therefore, there are many different choices for the production set, and for the filter set. All these choices exist for each join considered by the System R optimizer, and there are an exponential number of these joins considered. Even after choosing some production set and some filter set, the cost of access of the inner relation may not be simple to compute. The filter set usually has to be joined with the inner relation, and the cost of this join has to be computed (computing this cost can be expensive if the inner relation is a view).

Our goal is to add a Filter join method to existing optimizers without affecting their asymptotic complexity. Such a large space of choices cannot be searched without significantly degrading the performance of the optimizer. Therefore, our next task is to limit the search space to some tractable size.

### 3.3 Limiting the Search Space

We will now apply limitations on the search space for Filter joins. Recall that the search space is large because of the many possible choices of the production and filter sets.

The first limitation concerns the production set. The production set could in principle be composed from any subset of $R_{most}$. However, if it does involve some relation $R_{k+j}$ which is not part of the composite outer relation of the join, that relation will be accessed once again later in the join order. This implies that the relation $R_{k+j}$ will be joined at least twice! We would like to avoid such a situation. Similarly, if the production set does not correspond to some prefix of the outer relation, some joins will have to be repeated. For example, a production set of the form $\{R_1, R_3\}$ causes repetition of joins, while $\{R_1, R_2\}$ does not. Therefore, we have:

**Limitation 1:** *We require that the production set be composed of some prefix (i.e subplan) of the outer relation of the join. This guarantees that no joins will need to be repeated because of the Filter join method.*

There are still a number of possible prefixes of the outer relation ($k - 1$ choices in our example).

- If all of these choices were attempted, the complexity of optimization would be increased by a factor of $O(N)$ (given $N$ relations in the join).
- Further, the optimizer needs to be able to determine the cost and statistics of each plan prefix. While these have been computed at some point in the bottom-up plan generation, they are not necessarily stored. For example, when a System R optimizer is generating joins of four relations, it only needs to maintain the best plans for joins of three relations; costs and statistics for two-way joins are typically no longer available.

If one is willing to incur the increase in complexity, and if the particular optimizer implementation maintains

| | |
|---|---|
| $JoinCost_P$ | Cost of performing the joins required to generate production set P. |
| $ProductionCost_P$ | Cost of materializing production set P. |
| $ProjCost_F$ | Cost of projecting P to generate the filter set F. |
| $AvailCost_F$ | Cost of making the filter set F available to the inner relation $R_k$. |
| $FilterCost_{R_k}$ | Cost of generating $R_k$ and restricting it using the filter set F. |
| $AvailCost_{R_{k'}}$ | Cost of making $R_{k'}$ available to be joined with the outer relation. |
| $FinalJoinCost$ | Cost of performing the final join of the outer relation and and $R_{k'}$. |

Table 1: Cost Components of a Filter Join

the desired information, Limitation 2 is not required. However, if either of these is a concern, we can strengthen Limitation 1 as follows:

**Limitation 2:** *We require that the complete outer relation be treated as the production set.*

The next limitation deals with the filter set choices. Typically, the number of join attributes in any specific join is small, so that all possible alternative filter sets may be quickly searched by the optimizer. If this is not the case, we can require that all available join attributes be used in the filter set. Similarly, the number of implementation techniques (relations, Bloom filters, etc) is usually small. These details are usually implementation-specific; therefore we state the third limitation very broadly:

**Limitation 3:** *We require that some small and constant number of filter sets will be considered.*

Finally, let us make the following important assumption, that will be explained in the next section.

**Assumption 1:** *We will assume that only $O(1)$ complexity is required to estimate the cost of executing the Filter join, and to estimate the cardinality of its result.*

Given the last assumption, if all three limitations are applied, then there is no change in the asymptotic complexity of join optimization, although the Filter join is being considered as an option. For each particular join considered, the Filter join method examines only one production set, a small constant number of filter sets and therefore, the cost estimation happens in constant time.

# 4 Cost Estimation for a Filter Join

We now derive a formula to capture the costs of the proposed Filter join algorithm. This is an important part of the paper, since it is here that the costs of various aspects of the join algorithm are specified.

Assume that the Filter join whose cost is being estimated has $(R_1 \bowtie R_2 ... \bowtie R_{(k-1)})$ as the outer relation and virtual relation $R_k$ as the inner relation. Note that because of the limitations, the production set is simply the outer relation. The join evaluation cost may be broken up into the components as shown in Table 1 and explained below. The total cost of the Filter join is the sum of these seven costs.

$$JoinCost_P + ProductionCost_P + ProjCost_F + AvailCost_F + FilterCost_{R_k} + AvailCost_{R_{k'}} + FinalJoinCost$$

We now explain how each of these costs can be derived[2]:

[2]Note that the cost formula as described includes the cost $JoinCost_P$ of computing the outer relation.

$JoinCost_P$: When Limitations 1 and 2 are applied to reduce the search space, the production set $P$ is simply the outer relation. $JoinCost_P$ is therefore the cost of the outer relation, which is already computed as part of the bottom-up algorithm.

$ProductionCost_P$: $P$ needs to be materialized because it is used both in the generation of the filter set, and also in the top-level join. The cost of materializing the production set is a simple function of the cardinality of $P$. Since this cardinality is known (i.e already estimated by the optimizer), the materialization cost may be computed. Instead of creating a temporary relation, $P$ could also be recomputed. The cost of recomputation of $P$ is the same as $JoinCost_P$. Whichever cost is lower (materialization or recomputation) is chosen as $ProductionCost_P$.

$ProjCost_F$: The cost of performing a distinct projection of $P$ depends on the cardinality of $P$. It also depends on whether $P$ is sorted or not (a property which is maintained by the optimizer as "interesting") and whether the projection can be combined with the generation of $P$. The optimizer usually has all the necessary information to make an estimate of the $ProjCost_F$.

$AvailCost_F$: $AvailCost_F$ is the cost of making the filter set available to the relation $R_k$, and this depends directly on the size of the filter set. This usually involves materializing $F$. If the generation of $F$ can be pipelined with its use in restricting the inner relation, this cost is zero. If Bloom-filters are used to implement the filter set, the size of the filter set is fixed. If the filter set is stored as a relation, however, its size needs to be estimated. While we note that it is notoriously difficult to estimate the cardinality of projections [HOT88, LNSS93], the optimizer can make a estimate based on the cardinality of the production set $P$, and assumptions about the distributions of values in the various columns [Yao77]. Traditional optimizers already use some approximate technique to estimate projection cardinality; any such available technique can be used.

$FilterCost_{R_k}$: This is the cost of restricting $R_k$ using the filter set $F$. As per Assumption 1, we assume that this cost can be determined with $O(1)$ complexity, and that the optimizer can determine the cardinality of the $R_{k'}$ relation (i.e. after $R_k$ has been filtered by the filter set) with $O(1)$ complexity. We shall discuss these assumptions in detail in the next section.

$AvailCost_{R_{k'}}$: This is the cost involved if $R_{k'}$ needs to be materialized to be joined with the outer relation. This is a simple function of the cardinality of $R_{k'}$. Once again, if the generation of $R_{k'}$ can be pipelined with its use in the subsequent join, this cost is zero.

$FinalJoinCost$: This is easy to compute. The cardinality of the outer relation is known. The cardinality of the filtered inner relation has just been computed. The cost models of other unindexed join methods can be applied to determine the cheapest way to execute the final join.

The total join cost is the sum of the individual costs specified above. The only portion of this cost formula that has not yet been explained is the determination of $FilterCost_{R_k}$ (described in the next section).

## 4.1 Justifying Assumption 1

We assumed that the cost of filtering $R_k$ could be determined in constant time, along with the cardinality of the result. Since $R_k$ can be a complex relational expression involving several joins, this may require a nested invocation of the plan optimizer to plan the evaluation of $R_k$ along with its filtering by the filter set. Let the cost of the nested invocation of the optimizer for the operation of filtering $R_k$ be $C$.

If $C$ is a small constant, this is an acceptable complexity. This is the case when $R_k$ is a single relation. Determining the best plan for restricting $R_k$ with the filter set is simply a matter of applying a few simple formulas for well-known filtering methods (selection via Bloom filter, or join with magic set).

If $R_k$ involves a significant amount of computation however (for instance, if $R_k$ is a table expression involving many joins), the complexity $C$ of optimizing it may be significant. This is the typical case when the inner relation is a view in SQL, since the exponential-time optimization algorithm must be invoked on a non-trivial query of multiple relations.

At an abstract level, what is needed is a *parameterized* plan for the restriction of $R_k$. The basic idea is to treat the parameterized plan as a function whose parameter is the filter set. The result of invoking the function with the specific parameters is a particular instance of the plan. We would like to be able to generate the parameterized plan just once. Each specific instance of the plan is obtained at a cheap constant cost of instantiating the parameterized plan with the actual instances of the parameter. The specific plan instance would provide the cost of the restriction, as well as the cardinality of the result. Parameterized query optimization has been the topic of recent research interest, and there have been encouraging initial results [INSS91]. However, current techniques are still preliminary; consequently, we need a more concrete technique to deal with parameterization in the context of our query optimization problem.

## 4.2 Approximate Optimization of the Inner Restriction

Ideally, we would like to determine a precise query plan for the restriction of the virtual inner relation by the filter set. This is a non-trivial task. If the virtual relation is complex (e.g. a view with an aggregate function), even a simple textual magic rewriting of it can be tedious to implement. Instead, we propose the following simple approximation of parametric planning that can be used for the specific case of optimizing the restriction of $R_k$ by a filter set. The approximation is based on these general principles:

- Instead of actually generating a plan for the inner virtual relation, we need only generate its cost and cardinality. This information is sufficient for the join optimization to continue.
- When a final choice of the join order and join methods is made, the choices can be used to specify the SIPS for the existing rewrite implementation of magic sets. The actual plans for the sub-components can be generated after the magic rewriting is performed.

So our problem reduces to approximating the cardinality of the inner relation after being filtered by a particular filter set, and the cost of using the filter set to restrict the inner relation. One way to approximate parametric functions is to classify the actual parameter values into equivalence classes, so that any two instantiations of the parameters in the same equivalence class share the same function result. Therefore, there need be only as many invocations of the optimization algorithm as there are equivalence classes. Whenever the cost (or cardinality) is required for a particular filter set, if the corresponding equivalence class for the filter set has already been explored, its results are available and can be retrieved in constant time. If the desired result can be determined by using results for other equivalence classes (by extrapolation, for instance), then this is also a constant time computation. Otherwise, the desired result must be computed and stored in the table.

The choice of equivalence classes is important. The greater the number of equivalence classes, the more the complexity involved, but of course, the greater the accuracy of the cost estimates. This provides a performance "knob" with which a database system can modify the tradeoff between optimization cost and accuracy (and hence the potential for execution improvements).


Figure 4: Function for Filter Selectivity

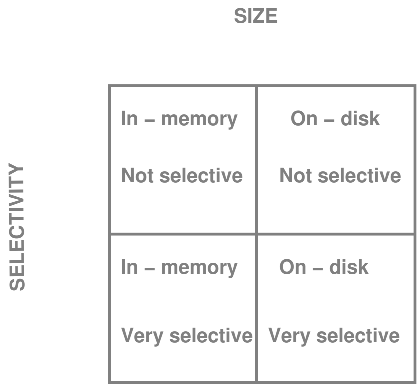


Figure 5: Function for Filter Cost

For the particular problem at hand, we still need to specify what the equivalence classes are, and how the result table is searched.

We first make the observation that the cardinality of the result of the filtered inner relation is directly proportional to the *selectivity* of the filter set, and does not depend on the set's physical size or implementation. Two instances of the parameters are in the same equivalence class if they have the same filter set selectivity. Once the selectivity has been computed for a few equivalence classes (i.e. for a few values of the cardinality of the filter set), a straight line can be fitted to them, thereby defining the selectivity for all other equivalence classes. We demonstrate this graphically in Figure 4. It is admittedly a heuristic decision to use a straight line fit. Other heuristic choices could be made instead.

Estimating the execution cost requires knowledge of the filter set selectivity and implementation. Once again, heuristics need to be used to determine how to create equivalence classes. For instance, it may be reasonable to consider only four equivalence classes as shown in Figure 5. The implementation of the filter set may be related to its size; this allows us to ignore one of the parameters. As in the previous case, other heuristics could be used to generate equivalence classes.

At this stage, we have described how a Filter join can be added to a cost-based optimizer. While it may appear at first glance that we make many assumptions in determining the cost and cardinality, the reader should realize that all stages of query optimization make a number of approximations in costing and cardinality estimation. Further, while our estimates are admittedly approximate, they are better than no estimate at all (which is the current state of the art with respect to algorithms like magic sets). That is, using these techniques is very likely to improve the results of today's query optimizers.

# 5 Filter Join in Other Domains

The Filter join is a useful join method for queries involving virtual relations other than views or table expressions. In every domain, the optimizer issues are the same, and the solutions similar. This section illustrates the applicability of the work to different domains.

## 5.1 Remote Joins in a Distributed/Heterogeneous DBMS

The Filter join method used in distributed databases is much better known as a semi-join. In a distributed DBMS, a semi-join has been proposed as an implementation level join algorithm that reduces communication costs at the expense of some added local processing costs. Heuristically, it is expected to improve performance.

Let us consider an example of a join between relations A at $Site_A$ and B at $Site_B$ in a distributed DBMS (well-known examples of such systems are System R* [LMH+85] and SDD-1 [BGWRR81]). Assume that the join needs to be performed at the site of the outer relation, i.e. $Site_A$.

When the join is performed using a semi-join technique, a distinct set of the join values of A is collected as a filter set which is shipped to $Site_B$. There it is used to filter B (i.e. it is joined with B), and the result is shipped back to A to be joined with A. This is the default join evaluation method in SDD-1. This algorithm can be effective when the filter set is small, and when the filter set is very selective (i.e. it filters out much of B). One could also use some superset of the filter set that can be stored more compactly, at the cost of some loss of selectivity. This is the idea behind the Bloom filter, which is a fixed size bit vector representing a superset of the filter set.

In SDD-1, semi-joins were the only join method, based on the assumption that communication costs were much greater than local processing costs. Any method that minimized communication overhead was therefore supposed to be the most efficient, and the semi-join was chosen for this reason. In fact, the join order determination was based on a ranking of the various possible semi-joins on the basis of communication cost. Critics of semi-joins argue that this technique requires that the outer relation be accessed twice, once to generate the filter set, and once to compute the final join [ML85]; this adds to the local processing costs. Therefore, in the System R* optimizer, semi-joins were not considered, based on the assumption that they added too much to local processing costs. In reality, both local and communication costs can be important, and their relative importance should be captured by appropriate cost metrics. If the remote relation is a view, it becomes even more important to correctly cost the various alternatives and make the right choice.

The cost formulas presented in the previous section need minimal modification to be adapted to the distributed DBMS domain. The cost $AvailCost_F$ of making the filter set available to the inner relation now has to take into account the cost of shipping it to the site of the inner. The cost $AvailCost_{R_k'}$ now has to take into account the cost of shipping the filtered inner relation back to the site of the outer relation. The rest of the cost formula is unchanged.

## 5.2 User-Defined Relations

Current object-relational systems use another kind of virtual relation; these are *user-defined relations*, which are generated by functions defined in a programming language. User-defined functions and methods are special cases of virtual relations that contain a single tuple for each specific set of argument values. The functions are typically invoked repeatedly with different argument values. Optimization research has focused on choosing the position within the plan at which these repeated invocations should occur [HS93]. The function results can also be cached to avoid duplicate invocations [HS93]. In this domain, it certainly is possible to investigate Filter Join as a join method. This will result in a number of function invocations being executed consecutively, resulting in possible benefits of locality inside the function computation. There will be no duplicate function invocations, because of the elimination of duplicates in the filter set. In current systems, such an option is not considered by the optimizer.

### 5.3 Filter Joins with Non-Virtual Relations

Filter joins can even be efficient for joins involving simple stored relations. The join algorithms commonly considered for local materialized relations are nested loops, hash join, sort merge and variants of these. However, a local semi-join could be a more efficient join method. Assume that the filter set is small enough to fit in memory. It can be created in a single scan of the outer relation. This can then be joined with the inner relation. Since the filter set is in memory, the join of the filter set with the inner can be computed in a single scan of the inner. Finally, the result needs to be joined with the outer relation. If the result fits in memory as well, this requires only a single scan of the inner. So in certain situations, the join can be performed with two scans of the outer and one scan of the inner, which may be much cheaper than any of the other join methods. In fact, in Starburst, semi-joins were added as a join method [HCL+90]; however, the purpose was to prove the extensibility of the optimizer, not to study the optimization issues involved in adding a Filter join method.

## 6 Related Work

Magic sets rewriting was originally used in the area of recursive query processing in deductive databases [BR91]. The impact of different choices if SIPS has been discussed in [BR91], and the idea of using approximations of the magic set has been explored in [Sag90, SS88]. Magic sets was shown to be applicable to complex SQL queries [MFPR90, SPL94], and was implemented in the Starburst database system [MP94]. In all implementations, magic sets has been treated as a query rewrite. Our interpretation of magic sets as an instance of a Filter join method distinguishes our current work from this earlier work, and allows us to integrate magic sets with a cost-based optimizer. In [SSS95] which was developed independently at the same time as our work, magic sets rewriting is described as being a "generalized semi-join". The idea is similar to ours in that magic sets has effectively been reduced to an algebraic operator. The authors then specify a number of algebraic transformations that can be used by a rule-based optimizer to find equivalent algebraic representations of a query involving such a generalized semi-join. Our work differs because it treats magic sets as a join method, because we have specified detailed cost formulas for this method, and because we explore the search space in the same manner that joins are explored.

In the literature on distributed databases, there has been extensive research on semi-join techniques. Some of this was discussed earlier in the paper. The work on semi-joins assumed that relations were simple stored relations, and therefore the costs of performing the semi-joins could be easily computed. Further, issues like the choice of SIPS were not considered, usually because communication costs were assumed to outweigh local processing costs (consequently, the chosen semi-join was always as restrictive as possible). Instead, optimization focused on the correct order in which to execute the semi-joins [BGWRR81]. The literature on heterogeneous databases has not yet dealt with issues like remote views in a complex query. However, such issues are becoming increasingly important, and our work should be applicable to this domain.

While researchers have long recognized the similarity between semi-joins and magic sets rewriting, this paper represents the first time that common optimization techniques are presented for both of them. This is also the first time that enhancements used for one technique have been considered for use in the other. For example, the possibility of implementing magic sets using a representation like a Bloom filter was not previously considered. These are the advantages of treating these and other techniques as instances of a common Filter Join method.

# 7 Conclusion

To summarize this paper:

- We have introduced the notion of virtual relations to model various kinds of relations that are especially important for query optimization.
- In previous work, variants of our Filter join algorithm have usually been applied as heuristics, and have not typically been tightly integrated with an optimizer. We have analyzed the challenges presented by this join algorithm, and have presented a practical mechanism to add it to existing optimizers without changing the asymptotic complexity of optimization.
- This contribution can result in more efficient evaluation of joins in decision-support SQL applications, in queries spanning distributed or heterogeneous databases, and possibly in other domains as well.

# References


[BGWRR81] Philip A. Bernstein, Nathan Goodman, Eugene Wong, Christopher L. Reeve and James B. Rothnie. Query Processing in a System for Distributed Databases (SDD-1) *ACM Transactions on Database Systems*, Vol. 6, No. 4, pages 602-625, December 1981.

[BR91] Catriel Beeri and Raghu Ramakrishnan. On the Power of Magic. *Journal of Logic Programming*, 10:255, 1991.

[GHK92] Sumit Ganguly, Waqar Hasan and Ravi Krishnamurthy. Query Optimization for Parallel Execution. In *Proceedings of ACM SIGMOD '92 International Conference on Management of Data, San Diego, CA*, 1992.

[HCL+90] L. Haas, W. Chang, G.M. Lohman, J. McPherson, P.F. Wilms, G. Lapis, B. Lindsay, H. Pirahesh, M. Carey, and E. Shekita. Starburst Mid-Flight: As the dust clears. *IEEE Transactions on Knowledge and Data Engineering*, March 1990.

[HOT88] Wen-Chi Hou, Gultekin Ozsoyoglu and Baldeao K. Taneja. Statistical Estimators for Relational Algebra Expressions. In *Proceedings of the Seventh Symposium on Principles of Database Systems (PODS)* pages 276–287, 1988

[HS93] Joseph M. Hellerstein and Michael Stonebraker. Predicate Migration: Optimizing Queries With Expensive Predicates. In *Proceedings of ACM SIGMOD '93 International Conference on Management of Data, Washington, DC*, pages 267–276, 1993.

[IK84] Toshihide Ibaraki and Tiko Kameda. Optimal Nesting for Computing N-relational Joins. In *ACM Transactions on Database Systems*, Vol.9, No.3, pages 482–502, October 1984.

[INSS91] Y. Ioannidis and R. Ng and K. Shim and T. K. Sellis. Parametric Query Optimization. In *Proceedings of the International Conference on Very Large Databases (VLDB)*, pages 103–114, 1992.

[KBZ86] Ravi Krishnamurthy, Haran Boral, and Carlo Zaniolo. Optimization of Nonrecursive Queries. In *Proceedings of the Seventeenth International Conference on Very Large Databases (VLDB)*, pages 128–137, 1986.

[Kim82] W. Kim. On Optimizing an SQL-like Nested Query. *ACM Transactions on Database Systems*, Vol.7, No.3, September 1982.

[LMH+85] Guy M. Lohman, C. Mohan, Laura M. Haas, Dean Daniels, Bruce G. Lindsay, Patricia G. Selinger and Paul F. Wilms. Query Processing in R*. In *Query Processing in Database Systems*, (W. Kim, D.S. Reiner, and D.S. Batory, eds.), Springer-Verlag, pages 30-47, 1985.

[LNSS93] Richard J. Lipton, Jeffrey F. Naughton, Donovan A. Schneider and S. Seshadri. Efficient Sampling Strategies for Relational Database Operations. *Theoretical Computer Science*, No.116, pages 195-226, 1993.

[MFPR90] Inderpal Singh Mumick, Sheldon Finkelstein, Hamid Pirahesh, and Raghu Ramakrishnan. Magic is Relevant. In *Proceedings of ACM SIGMOD '90 International Conference on Management of Data, Atlantic City, NJ*, 1990.

[ML85] L.F. Mackert and G.M. Lohman. R* Optimizer Evaluation. In *Proceedings of ACM SIGMOD '85 International Conference on Management of Data, Atlantic City, NJ*, 1985.

[MP94] Inderpal Singh Mumick and Hamid Pirahesh. Implementation of Magic-Sets in Starburst. In *Proceedings of ACM SIGMOD '94 International Conference on Management of Data, Minneapolis*, 1994.

[PHH92] Hamid Pirahesh, Joseph Hellerstein, and Waqar Hasan. Extensible/Rule Based Query Rewrite Optimization in Starburst. In *Proceedings of ACM SIGMOD '92 International Conference on Management of Data, San Diego, CA*, 1992.

[RSSS93] Raghu Ramakrishnan, Divesh Srivastava, S. Sudarshan and Praveen Seshadri. Implementation of the CORAL deductive database system. In *Proceedings of ACM SIGMOD '93 International Conference on Management of Data, San Diego, CA*, 1993.

[SI90] Arun Swami and Balakrishna R. Iyer. A Polynomial Time Algorithm for Optimizing Join Queries. Research Report RJ 8812, IBM Almaden Research Center, June 1992.

[SACLP79] Patricia G. Selinger, M. Astrahan, D. Chamberlin, Raymond Lorie, and T. Price. Access Path Selection in a Relational Database Management System. In *Proceedings of ACM SIGMOD '79 International Conference on Management of Data*, pages 23–34, 1979.

[Sag90] Y. Sagiv. Is there anything better than magic? In *Proceedings of the North American Conference on Logic Programming*, pages 235–254, Austin, Texas, 1990.

[SPL94] Praveen Seshadri, Hamid Pirahesh, and T.Y.Cliff Leung. Decorrelating Complex Queries. Research Report RJ 9846, IBM Almaden Research Center, 1994.

[SS88] Seppo Sippu and Eljas Soisalon-Soinen. An optimization strategy for recursive queries in logic databases. In *Proceedings of the Fourth International Conference on Data Engineering*, Los Angeles, California, 1988.

[SSS95] Divesh Srivastava, Peter J. Stuckey and S. Sudarshan. The Magic of Generalized Semijoins. Preliminary draft, ATT Bell Labs, Murray Hill, NJ, March 1995.

[SSW94] Konstantinos Sagonas, Terrance Swift and David S. Warren. XSB as an Efficient Deductive Database Engine. In *Proceedings of ACM SIGMOD '94 International Conference on Management of Data*, pages 442–453, 1994.

[Val90] Patrick Valduriez. Join Indices. In *ACM Transactions on Database Systems*, Vol.12, No.2, pages 219–246, June 1987.

[Yao77] S.B. Yao. Approximating the Number of Accesses in Database Organizations. In *Communications of the ACM 20,4*, pages 260–261, April 1977.

# A Cross-Applicability of Join Techniques

One of the contributions of this paper is to generalize magic sets, semi-joins, and similar techniques into a single Filter Join, which can be considered by a cost-based optimizer. This unifies techniques from different application domains. In Figure 6 below we summarize a large category of join techniques used in many different domains, and show how they are analogous. These analogies may be used to explore different ways of applying well-known techniques in new domains. In the table, any technique in a cell may be applicable in other cells in the same

| JOIN STRATEGY \ VIRTUAL RELATION TYPE | STORED RELATION IN A CENTR. DBMS | REMOTE RELATION IN A DISTR. DBMS | TABLE EXPR, SUBQUERY, VIEW | USER DEFINED RELATION |
|---|---|---|---|---|
| REPEATED PROBE | Indexed Nested Loops | Fetch Matches (System R*) | Correlation (nested iteration) | Procedure Invocation |
| w / CACHEING | | | | Function cacheing (memoing) |
| w / OUTER–SORT | Optimized Indexed Nested Loop | | Optimized nested iteration | |
| FULL COMPUTATION | Hybrid Hash, Sort–Merge Nested Loops | Fetch Inner, followed by local join (System R*) | Full decorrelation (Kim's method) | |
| FILTER JOIN | Local Semi–Join | Semi–Join (SDD–1) | Magic Sets | Consecutive procedure calls |
| LOSSY FILTER | | Bloom Filter | Magic Sets with partial SIPS | |

: NEW COST ESTIMATION

Figure 6: Join Techniques

thickly delineated row. For instance (as mentioned in Section 6) Bloom Filters could be used for subqueries instead of Magic Sets Rewriting.

We place every join technique into one of three categories, based on the way that the technique treats its inner relation. *Repeated probe* techniques probe the inner relation looking for matches to specific values found in outer tuples. *Full Computation* techniques require the entire inner relation to be available at once before they can be used. Filter Joins are described earlier in the paper, and require new cost estimation techniques.